PARALLEL DETECTION AND ELIMINATION OF STRONGLY CONNECTED COMPONENTS FOR RADIATION TRANSPORT SWEEPS

A Thesis

by

WILLIAM CLARENCE MCLENDON III

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2001

Major Subject: Computer Science

PARALLEL DETECTION AND ELIMINATION OF STRONGLY CONNECTED COMPONENTS FOR RADIATION TRANSPORT SWEEPS

A Thesis

by

WILLIAM CLARENCE MCLENDON III

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

December 2001

Major Subject: Computer Science

ABSTRACT

Parallel Detection and Elimination of Strongly Connected Components for Radiation Transport Sweeps. (December 2001)

William Clarence McLendon III, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Lawrence Rauchwerger

Discrete ordinate methods are commonly used to simulate radiation transport for fire or weapons modeling. The computation proceeds by sweeping the flux across a grid. A particular cell can not be computed until all the cells immediately upwind of it are finished. If the directed dependence graph for the grid cells contains a cycle, then sweeping methods will deadlock. This can happen in unstructured grids and time-stepped problems where the grid is allowed to deform. We describe a parallel algorithm to detect and break these cycles present in the directed dependence graphs of these grids as well as an implementation and experimental results on shared and distributed memory machines.

ACKNOWLEDGMENTS

I would like to thank my adviser, Lawrence Rauchwerger for the critical help he has provided during the duration of my master's work. I also thank Nancy Amato, Marvin Adams, and the rest of the ASCI group at Texas A&M for useful discussions regarding this work. Also, I wish to thank Bruce Hendrickson, Steve Plimpton, and Kent Budge at Sandia National Laboratories; without their guidance this work would not have been accomplished.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Detailed multi-physics simulations are computationally expensive problems and thus require enormous computational resources, if they are to be executed in practical time. Such large computational platforms usually consist of distributed parallel systems which have to execute the codes in fully parallel mode to ensure scalable performance. In this thesis we will consider a prototypical radiation transport solver used in an ASCI multi-physics code, such as SnRad [11] from Sandia National Laboratories. In this module the transport equations are solved using a sweep method. Sweep methods used in radiation transport discritize the radiation field by angle, and flux propagation is computed for a set of discrete directions or ordinates. The computation for each angle is performed by sweeping the flux across a grid, i.e., a finite element mesh commonly used for fluids or shock hydrodynamics modeling. Radiation enters a mesh cell via faces whose outward normals point upwind, and exits through downwind faces. This implies an order of computation on the grid cells which, for a single ordinate direction, is represented as a directed dependence graph. Two example meshes and their associated dependence graphs for a particular angle are shown in Fig. 1.

Each of the (typically several hundred) ordinate directions induces an associated dependence graph. Sweeping methods will deadlock if any of the dependence graphs contains a cycle [11], such as the one in the dependence graph for the twisted mesh shown in figure 1-B. Such situations occur frequently in 3-D unstructured grids and

This thesis follows the style and format of the *IEEE/ACM Transactions on Networking.*

in multi-physics problems where the underlying “object” that was meshed deforms over time.

To avoid deadlock, cycles in the set of ordinate dependence graphs must be detected and broken before the sweep can be performed. For example, key edges from these cycles can be removed eliminating the cycles and the transport sweep could use data from a previous iteration. This would allow a sweep to execute to completion without a deadlock. Since the mesh elements (vertices of the dependence graph) are distributed across processors, we require a scalable parallel algorithm for cycle detection.

The number of cycles can be exponential in the number of vertices but the number of *strongly connected components* (SCCs) is at most linear in the number of vertices since a vertex is in at most one SCC. Therefore we are interested in finding all *SCCs* of a directed graph. A strongly connected component of a directed graph, $G = (V, E)$, is defined as a maximal set of vertices, $U \subseteq V$, such that for every pair of vertices u and v in U, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$ [4], where $u \rightsquigarrow v$ means a directed path exists from u to v.

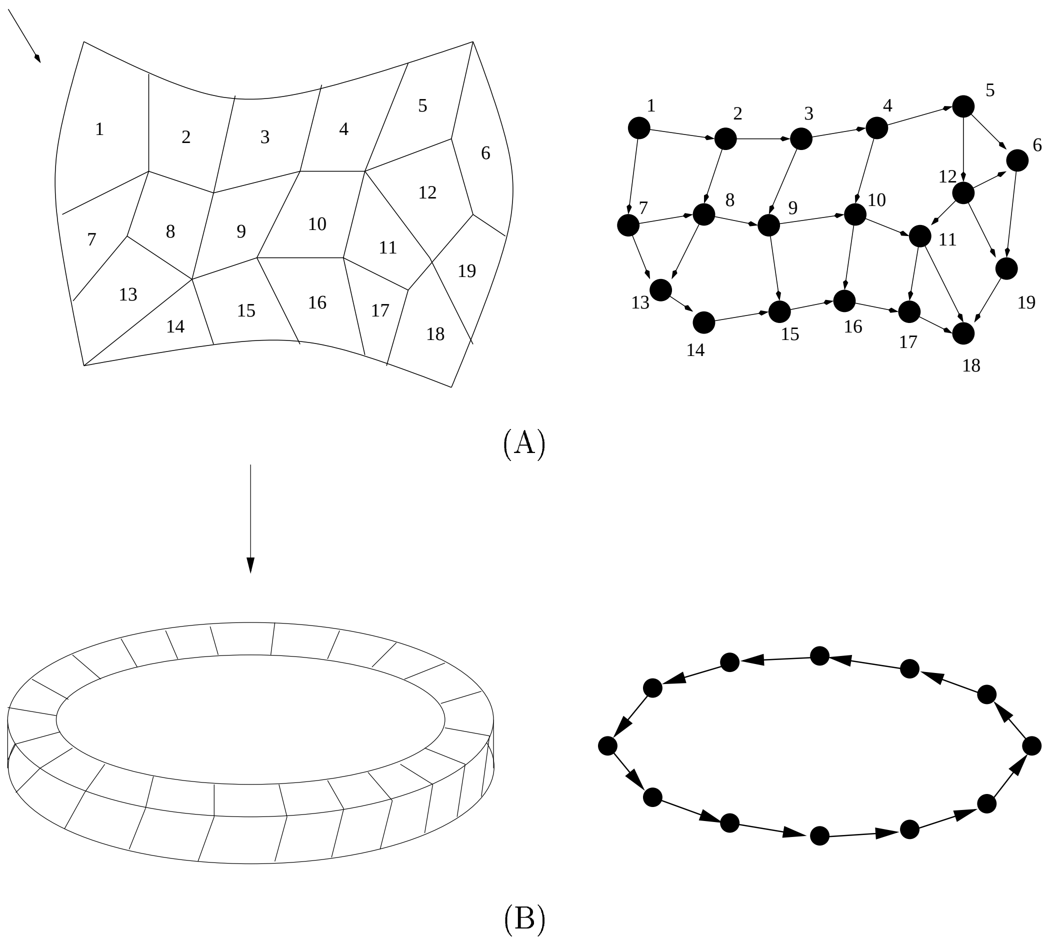


Fig. 1. Graph creation from meshes. (A) An unstructured finite-element mesh (left) and its associated acyclic dependence graph for the angle shown (right).
(B) A twisted ring of mesh elements that induces a cycle for the shown angle (left), and its dependence graph for the angle shown (right). A sweeping method will deadlock when it encounters a cycle such as this.

A. Previous Work

Tarjan's classic serial algorithm for detection of SCCs runs linearly with respect to the number of edges and uses depth-first search [13]. However, depth-first search is known to be difficult to parallelize. The special case of lexicographic depth first search is P-Complete [9; 12], which in practical terms means it is unlikely that a scalable parallel algorithm exists.

There are some parallel algorithms for detecting SCCs that do not rely on depth first search. Gazit and Miller have an NC algorithm which can be used for locating SCCs that uses matrix multiplication [6]. Vishkin and Cole [3] and Amato [1] have proposed optimizations or extensions of this algorithm, but they still require $O(n^{2.376})$ processors and $O(\log^2 n)$ time where n is the number of vertices in the graph. An NC algorithm developed by Kao for planar graphs was developed requiring $O(\log^3 n)$ time and $n/\log n$ processors [8]. Another efficient parallel algorithm for planar graphs is due to David Bader [2]. However our graphs arise from 3D finite element meshes and are non-planar. There are also some parallel algorithms for related problems in directed graphs [7; 10], but they are not well suited for our application either due to their complexity or because they do not directly compute SCCs.

B. Outline of Thesis

This thesis is organized as follows. In Chapter II we present the ModifiedDCSC algorithm for finding strongly-connected components. We also present a modification that allows the elimination of them via edge breaking. In Chapter III we describe our implementation of ModifiedDCSC to detect and eliminate SCCs for radiation transport sweeps on 3D unstructured meshes. We present optimizations made to the code specific to the radiation transport problem. Experimental results are presented in Chapter IV for various tests performed. Finally, conclusions are presented in Chapter V.

CHAPTER II

THE MODIFIED DCSC ALGORITHM

A. Detecting Strongly-Connected Components

The Divide-and-Conquer Strong Components (DCSC) algorithm of Fleischer *et al.* [5] is a divide–and–conquer approach for finding strongly connected components in a directed graph without using depth-first search. The main idea of DCSC is to recursively partition the directed dependence graph (DDG), $G = (V, E)$, so that all SCCs will be entirely contained within a partition. The recursion stops when partitions contain either single vertices or SCCs. The partitioning is based on the following Lemma [5]:

Lemma 1 *Let $G = (V, E)$ be a directed graph, with $v \in V$ a vertex in G, and let $Pred(G, v)$ and $Succ(G, v)$ denote the set of predecessors and successors of v in G, respectively. Then, the unique SCC containing v in G, denoted $SCC(G, v)$, if one exists, is $Pred(G, v) \cap Succ(G, v)$. Moreover, any SCC of G is a subset of $Pred(G, v)$, $Succ(G, v)$, or $Rem(G, v) = V - \{Pred(G, v) \cup Succ(G, v)\}$.*

The DCSC algorithm [5] initiates partitioning with a randomly chosen vertex $v \in V$, which we refer to as the *pivot*. The expected serial complexity of DCSC is shown to be $O(|V| \log |V|)$ when all vertices in G have constant degree. The meshes we are interested in have a bounded number of faces and therefore have a bounded number of edges as well, so this property holds.

The ModifiedDCSC algorithm we propose, outlined in Fig. 2, improves on the basic algorithm by performing a filtering or *trimming* step at the beginning of each recursive step which reduces the size of the graph that must be processed. In particular, *trimming* performs a topological traversal of G, and all vertices visited by this traversal are removed from G. Recall that a topological traversal begins from all vertices with in-degree zero, visits vertices after all their ancestors have been visited. It produces a linear ordering (a topological sort) of the vertices of G such that all edges are directed left to right. Thus, no vertices on a cycle, or vertices reachable from a cycle, will be visited by a topological traversal.

Algorithm: ModifiedDCSC(G)

```
1. IF $G$ is empty THEN return
2. trim() $G$ in forward direction
3. IF $G$ is not empty THEN
4.    trim() $G$ in backward direction
5.    Select pivot $v$ from the live vertices of $G$
6.    mark $Pred(G,v)$ and $Succ(G,v)$ in $G$
7.    $SCC(G,v) = Pred(G,v) \cap Succ(G,v)$
8.    DO in parallel:
9.       ModifiedDCSC( $Pred(G,v) - SCC(G,v)$ )
10.      ModifiedDCSC( $Succ(G,v) - SCC(G,v)$ )
11.      ModifiedDCSC( $Rem(G,v)$ )
12. ENDIF
```

Fig. 2. Algorithm ModifiedDCSC.

Trimming the graph is performed by the **trim()** routine in parallel, which is listed in Figure 3. We can perform this trimming in both the forward direction and reverse direction of the DDG simultaneously to achieve greater parallel efficiency.

In figure 4 the **mark()** routine is listed. It represents the DCSC phase of the ModifiedDCSC algorithm. Prior to the execution of mark(), the pivot vertex v is selected at random from G. Starting from v, mark() traverses G in breadth-first order in both forward and backward directions. It finishes when all the predecessors and successors of v have been visited and colored. A vertex is colored as predecessor or successor depending upon how it was reached during this traversal. Vertices visited by following a directed edge in the forward direction are colored as successors. Vertices visited by following an edge backwards are colored as predecessors.

Algorithm: trim()

```
INPUT :   DDG, $G$
OUTPUT:   DDG, $G$, with 0 or more vertices removed
1. push all vertices with indegree of 0 into work queue, $Q$
2. WHILE $terminate$ == false DO
3.    WHILE $Q$ is not empty DO
4.       pop a vertex $v$ from $Q$
5.       mark $v$ as dead
6.       FOR every child $u$ of $v$ DO
7.          IF $u$ is local THEN
8.             decrement indegree of $u$ by 1
9.             IF indegree of $u$ == 0 THEN push $u$ onto $Q$
10.         ELSE ($u$ is on another processor, $p_i$)
11.            Send information about $u$ to $p_i$
12.      ENDDO
13.   ENDDO
14.   IF there are messages waiting THEN
15.      Receive all incoming messages
16.      decrement indegree for every vertex received
17.      IF indegree == 0, push vertex onto $Q$
18.   ELSE
19.      $terminate$ = IsTerminated()
20. ENDDO
```

Fig. 3. Algorithm trim() in parallel.

Algorithm: mark

```
INPUT :   DDG, $G$
OUTPUT:  DDG, $G$, with vertices colored
1. FOR every pivot node, $v$, DO
2.    push $\{v$,forward$\}$ and $\{v$,backward$\}$ onto $Q$
3. WHILE $terminate$ == false DO
4.    WHILE $Q$ is not empty DO
5.       pop $\{v$,dir$\}$ from $Q$
6.       IF dir == forward THEN $v$.forward-mark = true
7.       ELSE
8.          $v$.backward-mark = true
9.       IF dir == forward THEN
10.         FOR every child $u$ of $v$ DO
11.            IF $u$ is local THEN
12.               IF $u$.forward-mark == false THEN
13.                  push $\{u$,forward$\}$ onto $Q$
14.            ELSE ($u$ is on processor $p_i$)
15.               Send $u$ and dir to $p_i$
16.      ELSE ($d$ == backward)
17.         FOR every parent $u$ of $v$ DO
18.            IF $u$ is local THEN
19.               IF $u$.backward-mark == false THEN
20.                  push $\{u$,backward$\}$ onto $Q$
21.            ELSE ($u$ is on processor $p_i$)
22.               Send $u$ and dir to $p_i$
23.         ENDDO
24.   ENDDO
25.   IF there are messages waiting THEN
26.      Receive all incoming messages:  $\{v$,dir$\}$
27.      IF dir == forward AND $v$.forward-mark == false THEN
28.         push $\{v$,forward$\}$ onto $Q$
29.      ELSE IF $v$.backward-mark == false THEN
30.         push $\{v$,backward$\}$ onto $Q$
31.   ELSE
32.      $terminate$ = IsTerminated()
33. ENDDO
```

Fig. 4. Algorithm mark() in parallel.

Based on Lemma 1, once G has been colored we can partition it into four regions:

Pred(G,v) - Vertices *from which* the pivot v can be reached along some path.

Succ(G,v) - Vertices that *can be reached* along a path from the pivot v.

Rem(G,v) - Vertices that are *neither* predecessors nor successors of v (the remainder). Notice that these vertices will not have been visited by *any* previous trim() or mark() yet in any previous recursive step.

SCC(G,v) - Vertices that are *both* predecessors and successors, the (unique) SCC containing v. $SCC = Pred(G, v) \cap Succ(G, v)$.

These partitions can be considered as independent graphs in terms of cycles. The vertices in $SCC(G, v)$ are removed from G and $Pred(G, v)$, $Succ(G, v)$, and $Rem(G, v)$ are recursively searched by ModifiedDCSC for additional SCCs.

In figure 3 line 19 and figure 4 line 32 there are references to a routine called *IsTerminated()*. This checks to see if the termination condition has been met. For both trim() and mark() to exit, each routine must meet the following exit conditions:

- No processor has any remaining work.
- No processor has any unreceived messages.

Termination detection adds overhead but it is required because we do not know beforehand how much of the graph will be traversed. The trim and mark routines may not visit all the nodes in the graph. In fact, unless the graph is acyclic the trim will be stopped at some point by a SCC.

The listings in figures 3 and 4 show that we loop until no more work remains locally, then we checks for incoming messages bringing work from an off-processor source. If additional work is picked up a processor will resume processing locally.

Once there is no more local work and there are no messages bring incoming work, we check to see if the termination condition has been met. These routines will not exit unless the termination conditions are satisfied.

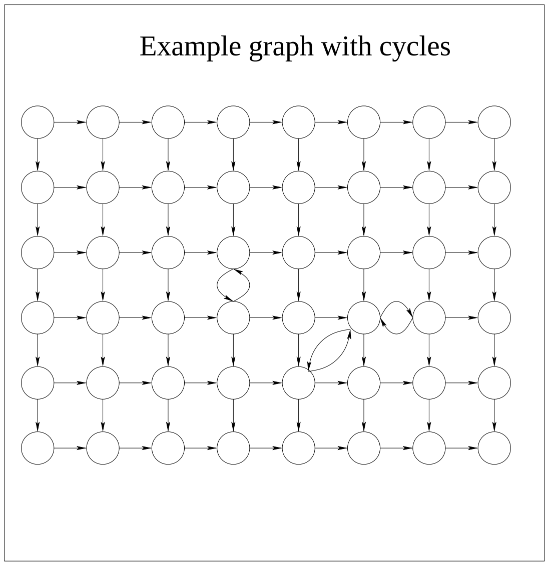
We can use different termination detection methods in these routines depending upon need and the machine architecture. For example, in a shared memory environment where all processors can “see” the whole address space, each processor can directly check the work queue to determine if any work remains globally. However, in a distributed memory environment each processor can see only its own local address space, and thus cannot read the status of other processors’ work directly. Processors must explicitly communicate their status in distributed memory so that all processors can know when to terminate. We used a token-passing scheme in our implementation and have found it to be adequate.

Figure 5 illustrates the execution of ModifiedDCSC on an example graph shown in panel A which contains two cycles. In Fig. 5-B, the effect of trimming is shown; vertices in the shaded region are removed by trim() in the forward and backward directions. In this example the entire graph cannot be ‘seen’ during the trim due to the blocking effects of the SCCs on trim(). After trim() terminates, the remaining graph will enter the DCSC phase of the code.

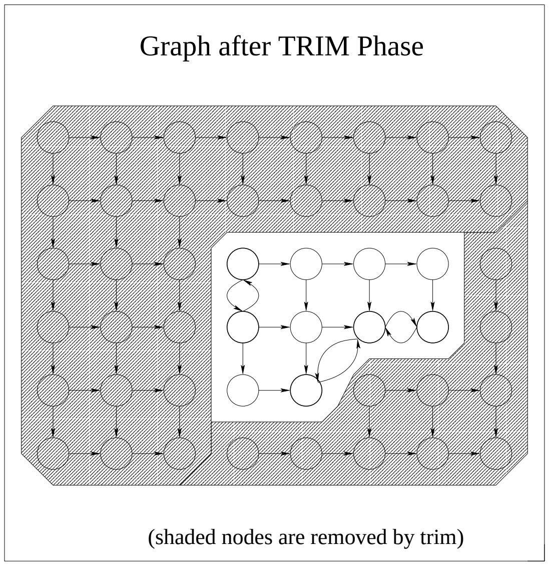
In the DCSC phase, a pivot vertex, v is selected as shown in Fig. 5-C, (a), as the shaded vertex. ModifiedDCSC then calls mark() to color the predecessors and successors of v as $Pred(G, v)$ and $Succ(G, v)$, respectively. After mark() finishes a strongly connected component, $SCC(G, v)$, is reported if found, and its vertices are removed from G.

The remaining vertices are partitioned according to their colors and considered as independent sub-graphs since, by Lemma 1 any remaining SCCs are wholly contained inside these partitions. Figure 5-C,(b), shows the coloring and partitioning of G after mark() has completed; in Fig. 5-C, (c), the SCC is shown as the nodes meeting the criteria in lemma 1. Finally, ModifiedDCSC may be recursively applied to the remaining sub-graphs of G, Fig. 5-C, (d).

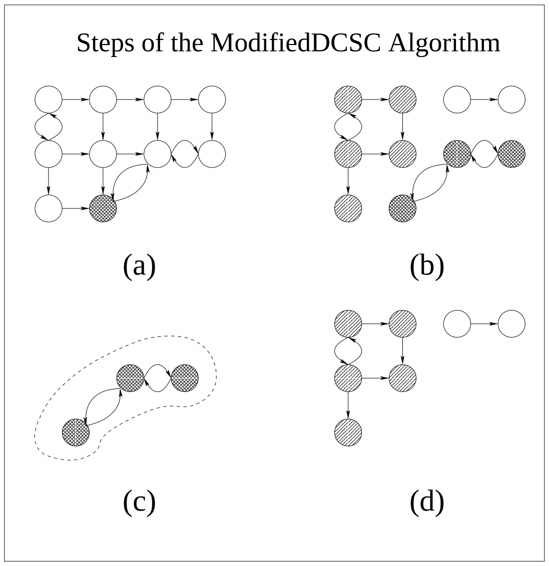
This example finishes with a second recursive step, shown in Figure 5-D. Vertices from the remaining partitions which are removed during trim() are shaded in (a). The only remaining vertices after trim() will be on the cycle, thus the pivot node selected prior to mark() will be part of the cycle. Finally, during mark() the each of the remaining vertices will be colored as both predecessor and successor, identifying the SCC. After mark() completes, the SCC is reported and removed from G. Since there are no longer any vertices left in G, ModifiedDCSC terminates and returns the two SCCs it found.



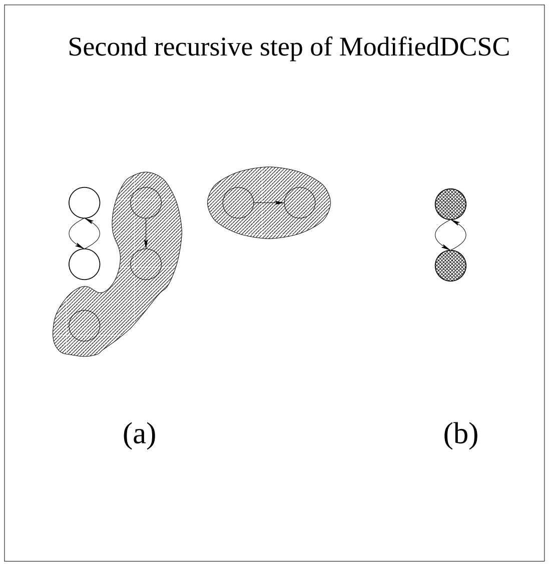
(A)



(B)



(C)



(D)

Fig. 5. ModifiedDCSC applied to an example graph.

B. Eliminating SCCs with ModifiedDCSC

We have now seen how ModifiedDCSC detects the SCCs in a graph in parallel. Recall that the motivation behind development of ModifiedDCSC was to enable radiation transport sweeps to work on unstructured 3D grids without deadlocking. To accomplish this, we must not only detect but also eliminate the SCCs from the DDGs. We can modify ModifiedDCSC to allow the elimination of SCCs from graphs by removing, or cutting, certain edges in G. The output of ModifiedDCSC can then include the list of SCCs found as well as a list of edges in G, which if broken will make G acyclic.

The listing in figure 6 illustrates the new ModifiedDCSC algorithm with our SCC elimination steps included. To remove SCCs, we insert an additional step in ModifiedDCSC after the detection of a SCC. Instead of removing $SCC(G, v)$ from G, we remove an edge from $SCC(G, v)$ and carry the SCC into the next recursive step as an additional partition of G.

If the edge broken removes the cycle, then trim() will remove the vertices in the SCC during the next recursive call to ModifiedDCSC. If removing the edge does not eliminate the SCC, then the next call to trim() during the next level of recursion will not fully eliminate the SCC. In this case some more vertices will be removed from the SCC and a new strongly connected component, SCC', will remain such that $SCC' \subseteq SCC$. This can happen for SCCs that are complicated with many cycles. We can continue removing an edge from the SCC with each recursive call to ModifiedDCSC until all of the cycles are gone and all vertices are removed by trim(). Often the SCCs contain single-cycles and will be eliminated by the first edge cut since a simple cycle can be broken by cutting any edge in the cycle.

Due to the divide-and-conquer nature of the DCSC method, removing the strongly connected components can be performed concurrently with the detection of new SCCs. This method allows SCC' to be considered as a fourth type of graph. SCC' is recursively searched in the same manner as the other partitions of G. Any SCCs found within SCC' during subsequent recursive calls are not reported by ModifiedDCSC because they have already been reported as part of SCC originally.

Edges broken via this process are reported in addition to the original SCCs found. The transport solver in a multiphysics application, such as SnRad [11], equipped with the knowledge of the SCCs and a list of edges that can be broken to allow a successful sweep can now be performed without deadlock by handling the cycles appropriately.

Algorithm: ModifiedDCSC_BreakSCC(G)

```
1. IF $G$ is empty THEN return
2. trim() $G$ in forward direction
3. IF $G$ is not empty THEN
4.    trim() $G$ in backward direction
5.    Select pivot $v$ from the live vertices of $G$
6.    mark $Pred(G,v)$ and $Succ(G,v)$ in $G$
7.    $SCC(G,v) = Pred(G,v) \cap Succ(G,v)$
7a.   SCC'(G,v) = SCC(G,v) - 1 edge
8.    DO in parallel:
9.       ModifiedDCSC_BreakSCC( $Pred(G,v) - SCC(G,v)$ )
10.      ModifiedDCSC_BreakSCC( $Succ(G,v) - SCC(G,v)$ )
11.      ModifiedDCSC_BreakSCC( $Rem(G,v)$ )
11a.     ModifiedDCSC_BreakSCC( SCC'(G,v) )
12.ENDIF
```

Fig. 6. Algorithm ModifiedDCSC_BreakSCC. Simple modification to ModifiedDCSC allowing SCC elimination by edge removal. Line 7a selects one edge from each SCC and removes it to create SCC'. Then in line 11a, we recurse on SCC' as a fourth partition.

CHAPTER III

IMPLEMENTATION

Our implementation of ModifiedDCSC is written in the C programming language and the Message Passing Interface (MPI) communications library. MPI was chosen because it performs well and it is portable across all parallel machines. This code is targeted for CPlant and ASCI Red at Sandia National Laboratories, both of which are massively parallel distributed memory platforms.

Most of the development was performed on a Hewlett Packard V-Class server at Texas A&M University. This machine is a 16-processor ccUMA SMP running 200 MHz PA-RISC processors.

Our implementation of ModifiedDCSC is optimized for the detection and elimination of strongly connected components occurring in DDGs resulting from 3D unstructured grids. Specifically we are interested in grids used by radiation transport calculations. This specialization allows several optimizations, which will be discussed in this chapter.

A. Constructing the Directed Dependence Graph

The multiphysics code uses a finite element mesh for its computation. We need to convert this mesh, M, into a directed dependence graph (DDG) for every ordinate vector. We briefly illustrated this construction in figure 1.

The method used to determine the orientation of each directed edge for every vertex is shown by the listing in figure 7. We also show a small example of how two adjacent mesh cells are changed into a graph with their edge directed according to an ordinate vector in figure 8.

Algorithm: CreateDDG

```
INPUT :   Finite element mesh $M$
          ordinate vector $\vec{d}$.
OUTPUT:  DDG, $G$.
1. FOR every cell, $u \subseteq M$ DO
2.    Add vertex $u$ to $G$
3. ENDDO
4. FOR every cell, $u \subseteq M$ DO
5.    FOR every face, $\hat{f} \subseteq u$ shared with adjacent cell $v$ DO
6.       $\vec{\eta}$ = outward face normal of $\hat{f}$
7.       IF $\vec{\eta} \cdot \vec{d} \geq \varepsilon$ THEN
8.          Add directed edge $\vec{uv}$ to $G$
11.      ENDIF
12.   ENDDO
13.ENDDO
```

Fig. 7. Constructing the DDG from an input mesh. The algorithm used to compute the DDG from the input mesh for each ordinate angle $\vec{d}$. ε represents an error tolerance for the dot-product computation.

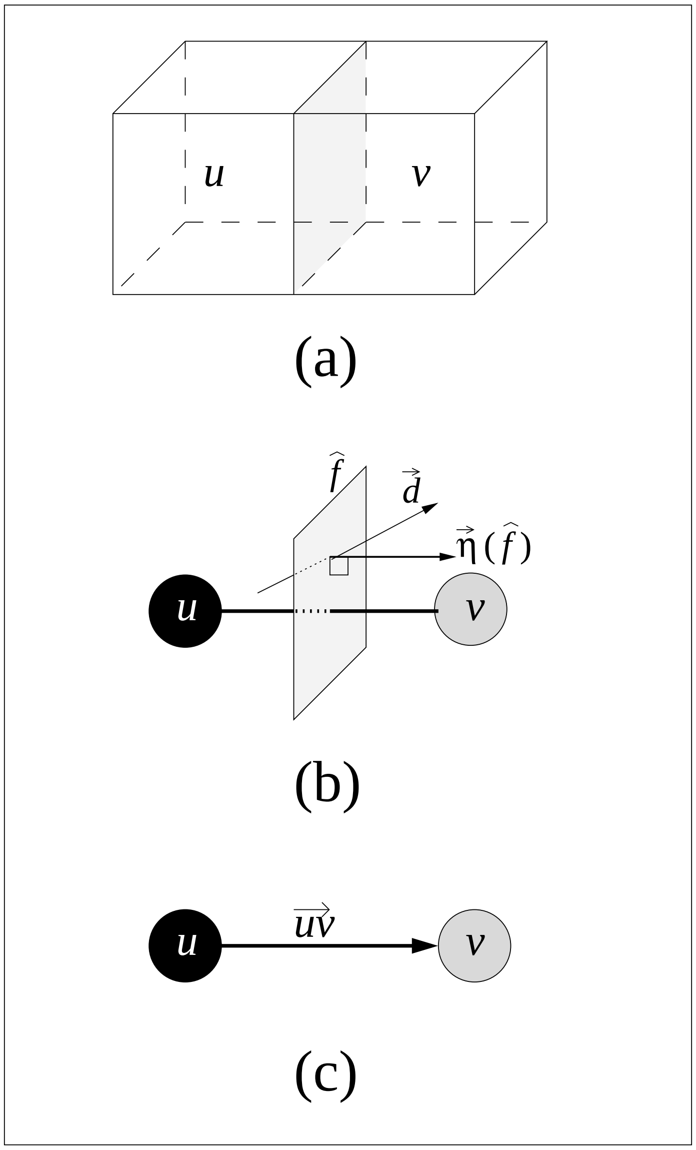


Fig. 8. DDG construction from a mesh. Construction of a directed dependence graph from a mesh. Adjacent cells u and v in (a) are represented as vertices u and v in (b). The shared face $\hat{f}$ represents an edge connecting u and v in the DDG. Edge $\overline{uv}$ is directed according to the relationship between the outward face normal $\vec{\eta}(\hat{f})$ of u. If the ordinate vector d makes an angle of less than 90 degrees with $\vec{d}$, then the edge is directed as $\vec{uv}$ (c). If d is orthogonal to $\vec{\eta}$ then there is no edge $\overline{uv}$.

B. Searching Over Many Ordinate Angles Simultaneously

Sweeping methods such as those commonly used in radiation transport involve a finite element grid being swept over a set of discrete ordinate angles. These ordinate angles can be visualized as starting from many points distributed in 3D space around the mesh. This 3D volume around the mesh is typically divided into 8 regions, called octants, which are divided by the x, y, and z axis planes.

A topological traversal of this kind is not fully parallel. It is limited to the length of the longest critical path between the starting vertices and the last vertex traversed. During each step along this critical path, available parallelism is limited to the number of vertices having an indegree of zero. The amount of available parallelism is dependent upon the characteristics of the input graph.

In our application every ordinate angle produces a DDG which is independent from the DDGs of other angles. Searching many DDGs simultaneously allows us to exploit additional parallelism because there are more vertices available at each step.

The DDGs are distributed in the same manner as the finite element mesh which they represent. Searching multiple angles simultaneously also allows many additional starting points for trim() since our angles are spread evenly in 3D around the mesh. This also increases the parallel efficiency of our implementation by getting more processors involved in the computation more quickly. Transport sweeps typically take advantage of this parallelism as well.

C. Taking Advantage of Paired Ordinates and Load Balancing

Another optimization related to radiation transport calculations which we can take advantage of is *ordinate pairing.* We say that two ordinates, $\vec{d_1}$ and $\vec{d_2}$ are *paired* if $\vec{d_1} = -\vec{d_2}$.

Recall from Chapter III Section A that an edge in the DDG is constructed for each cell face by comparing the outward face normal of the cell face to the ordinate angle. In the case where $\vec{d_1} = -\vec{d_2}$, *all* edges in $G(\vec{d_1})$ will be directed opposite of those in $G(\vec{d_2})$. There are also no additional edges added or removed between $\vec{d_1}$ and $\vec{d_2}$ as well. This means that an SCC found in $G(\vec{d_1})$ also exists in $G(\vec{d_2})$ because the *cycles* are preserved with their directed edges simply reversed.

ModifiedDCSC for radiation transport sweeps can take advantage of this fact by only searching *one* ordinate angle for every pair given in the input. When the ordinates are spread out evenly in 3D space and every ordinate is part of a pair. In that case, ModifiedDCSC only needs to search half of the actual ordinates given and reports the SCCs for both ordinates in each pair. This decreases the amount of work ModifiedDCSC is required to do by half, reducing the overall time to solution.

In our application, graphs are statically distributed and are not redistributed. The ordinate angle's relation to the mesh determines the starting vertex for trim(). We can achieve better performance when the angles are evenly distributed around the mesh. When selecting an angle from a pair, we pay attention to which octant the angles are in. We obtain better performance if the angles used are spread evenly over all 8 octants. The selection of angles for ModifiedDCSC is performed to keep the number of angles in each octant as equivalent as possible.

D. Storing the DDG

There are different ways the DDGs for our problem could be represented in a data structure. One method is to traverse the mesh directly, computing the edge directions each time a face boundary is traversed. The second method is to store a DDG for every ordinate angle directly in a more complete graph data structure, such as an adjacency list representation. Each method has its advantages and disadvantages.

Traversing the mesh directly uses much less memory and has better memory reuse than storing the graph. This becomes especially apparent when the input involves many hundreds of ordinate angles. Only one data structure is stored that represents the DDGs for all the input angles. This improves locality as well as using far less memory. There are some significant disadvantages to this approach which come directly from the recursive partitioning nature of the ModifiedDCSC algorithm. ModifiedDCSC can traverse an edge many times throughout the course of execution as the marking step finds the predecessors and successors of each subsequent pivot node. Computing the edge direction each time this occurs becomes expensive for problems in which there are many recursive steps performed.

If recomputing the edge direction with each traversal becomes too time consuming we can compute the edges once for every angle and store each one as a separate DDG. Storing the DDGs for all the angles increases the memory requirements for ModifiedDCSC. Though in the context of a radiation transport code which stores flux for every cell, angle and energy group this is not the dominant memory cost. The advantage of this is mostly in lowering the overall execution time by only computing the expensive dot product once per edge for every angle.

An early implementation of ModifiedDCSC adopted the first scheme in which we performed the SCC search directly on the mesh. Experiments run on ASCI Red showed better speedups, reaching 200+ on 256 processors. The execution time, however was observed to be significantly longer than when we precompute the edge direction and store the individual graphs. Because of this, in the current implementation we opt to store the complete graph using an adjacency list representation with ghost nodes to reduce overall execution time.

CHAPTER IV

EXPERIMENTAL RESULTS

We present experimental results obtained on a HP V2200 Exemplar server and on ASCI Red. The HP is a 16 processor ccUMA SMP machine maintained by the PARASOL laboratories in the Computer Science department of Texas A&M University. The processors are PA-RISC running at 200MHz. ASCI Red is a 9280 processor supercomputer maintained at Sandia National Laboratory. ASCI Red uses 333 MHz Intel Pentiums with a high bandwidth, low latency interconnection network.

We conducted experiments to show the impact of the addition of trim() to the DCSC algorithm. Experiments were also performed to test the scalability and performance of ModifiedDCSC with a variety of meshes as well as progressively deformed meshes. On both platforms, we use the same MPI distributed memory code without machine specific modifications.

A. Effectiveness of trim() in Elimination of Work

Our ModifiedDCSC algorithm [5] benefits greatly from the addition of the trim step. By trimming out nodes that can be easily determined as not part of any SCCs, the overall problem size is reduced. DCSC benefits from this reduced problem size because the set of possible vertices from which the pivot is selected is reduced, thus giving a higher probability of the pivot being part of a SCC.

We ran experiments to show the benefit of trimming the graph to the ModifiedDCSC method. Figure 9 and figure 10 illustrate a comparison of the total number of recursive steps taken and the amount of work (vertices + edges) at each step by ModifiedDCSC both with and without trim() enabled.

The mesh used for figure 9 is a deformed brick mesh, d-04, which is a $30 \times 30 \times 30$

cell brick mesh with corner nodes deformed to produce cycles. There are many SCCs of varying complexity and size well distributed throughout the mesh.

Figure 10 is executed on a mesh called s20, which represents the volume around a submarine hull. This mesh contains roughly 40,000 cells and has very few cycles. For s20, we expect that ModifiedDCSC will complete very quickly with few recursive calls.

We can observe that with trim() disabled, ModifiedDCSC will be called recursively many more times than if trim() is enabled. Also, we see that the percentage of vertices removed during each recursive step is much more when trim() is enabled, even in a graph with an artificially large number of SCCs.

The addition of trimming to ModifiedDCSC is a very practical improvement to the DCSC method. It results in a reduction of the number of iterations to solution as well as the amount of work per iteration. The raw execution time benefits from this improvement as well.

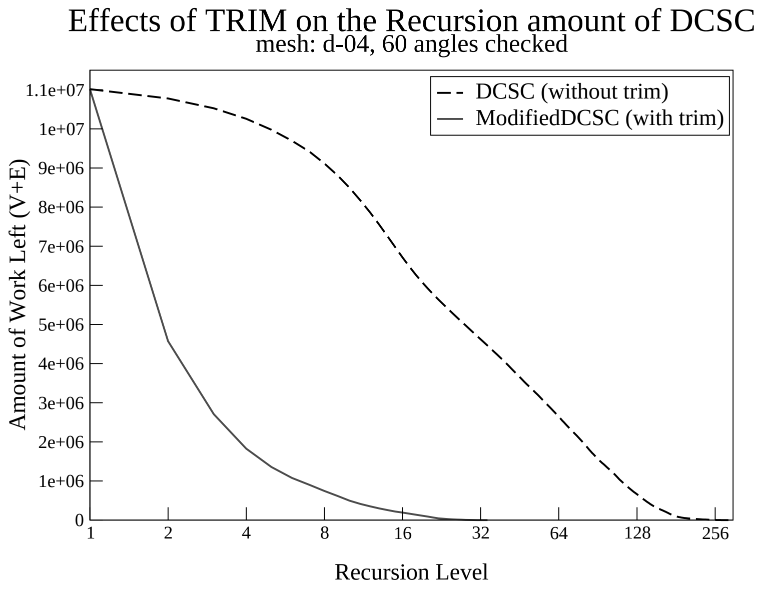


Fig. 9. Impact of trim() on a mesh with many SCCs.

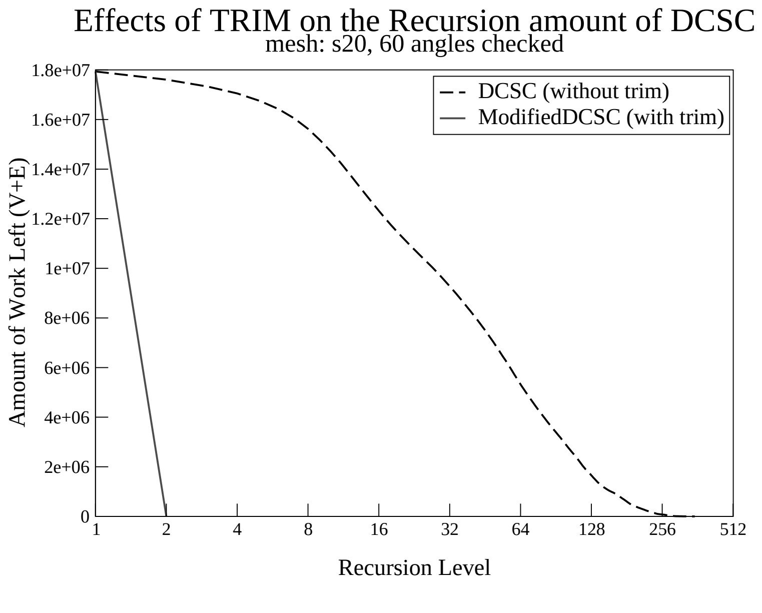


Fig. 10. Impact of trim() on a mesh with few SCCs.

B. Varied Geometries

The meshes listed in Table I are of different geometries, representing several different physical models we can use to test ModifiedDCSC. Tables II and III show the execution time and speedup achieved on ASCI Red for these meshes. The data shown in these tables are the results for a 120 ordinate problem which resulted in an actual search of 60 angles due to angle pairings.

Figures 11 and 12 show the scalability of ModifiedDCSC on these meshes for the HP-V2200 and ASCI Red to 16 and 64 processors, respectively. For these and subsequent speedup curves, we normalized against the single processor run time of ModifiedDCSC. In our experiments, the single processor ModifiedDCSC was usually at least as fast as Tarjan's serial algorithm for these problems.

Figure 12 on shows that an increased number of SCCs (b42000, b64000, and warpcyl) reduces scalability. As we have shown in Chapter IV, Section A, meshes with few SCCs benefit much more from trim(). Since BFS is not fully parallel, our parallel efficiency is expected to be better when the number of recursive steps is kept at a minimum.

Table I. Characteristics of meshes used in varied mesh experiments. These meshes were used to test ModifiedDCSC with some varied geometries.

Mesh	Description	Size	Angles Checked	Total SCCs	Avg. Size of SCC
b42000	Brick with many, evenly distributed cycles	42875	60	4420	4.7
b64000	Larger version of b42000	64000	60	17437	5.4
s20	Volume around a submarine hull Few, localized cycles	43984	60	4	20.0
sphere2	Solid spherical mesh. Few cycles near center	36712	60	1	8.0
warpcyl	Warped cylinder with concentric, stacked rings, many large cycles. Elements twisted by 18 degrees.	28000	60	280	400

Table II. Execution times for varied meshes on ASCI Red.

Varied Meshes on ASCI Red							
	Execution Time (seconds)						
Mesh	**1**	**2**	**4**	**8**	**16**	**32**	**64**
s20	37.3	19.1	9.8	5.3	2.9	1.6	0.9
sphere2	31.8	16.8	8.9	4.6	2.4	1.4	0.8
b42000	49.8	29.0	15.9	8.5	4.8	3.4	2.6
b64000	156.2	100.5	53.1	32.9	22.7	19.2	19.7
warpcyl	28.3	16.6	11.0	7.0	4.0	2.5	1.7

Table III. Speedups for varied meshes on ASCI Red.

	Speedup						
Mesh	**1**	**2**	**4**	**8**	**16**	**32**	**64**
s20	1.00	1.95	3.81	6.98	13.07	24.04	42.83
sphere2	1.00	1.89	3.56	6.85	13.07	22.86	40.73
b42000	1.00	1.71	3.12	5.85	10.30	14.59	19.36
b64000	1.00	1.55	2.94	4.75	6.88	8.14	7.93
warpcyl	1.00	1.71	2.56	4.02	7.10	11.49	17.03

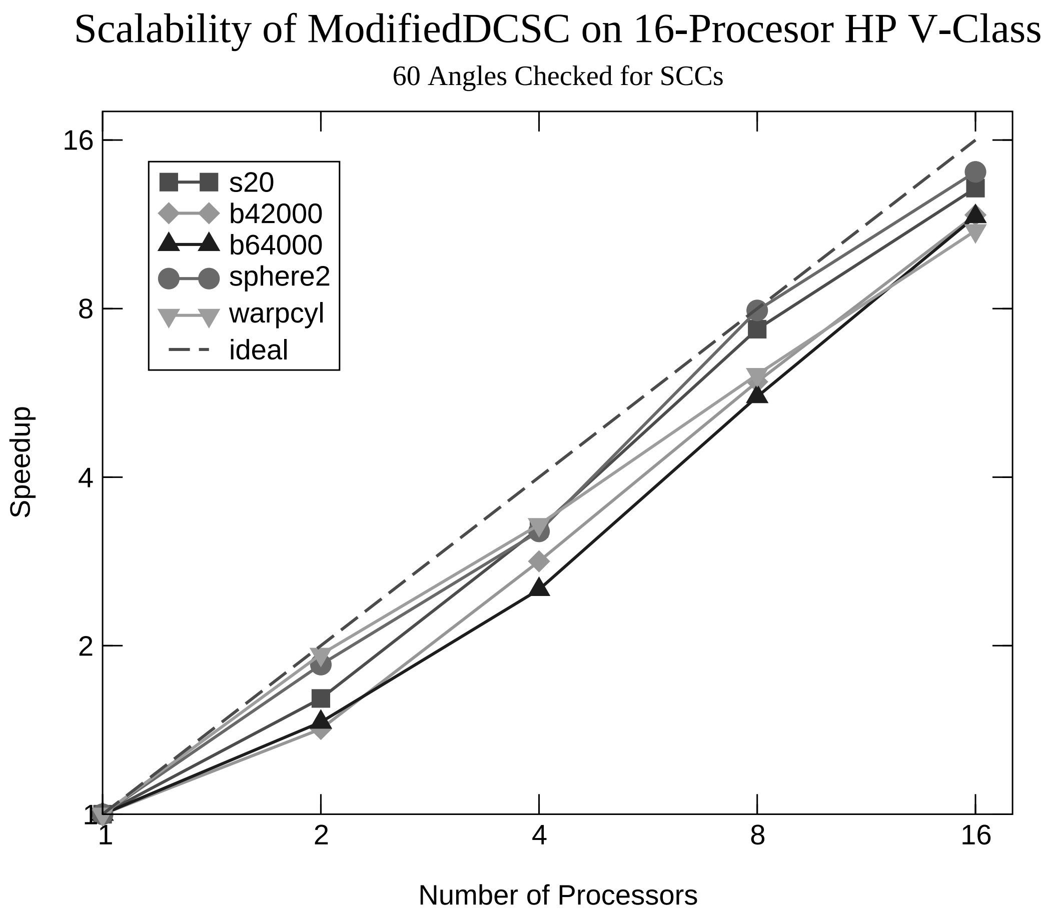


Fig. 11. Scalability of ModifiedDCSC on HP-V2200 for various meshes. This graph shows the scalability of ModifiedDCSC searching the DDGs of several meshes of different geometries.

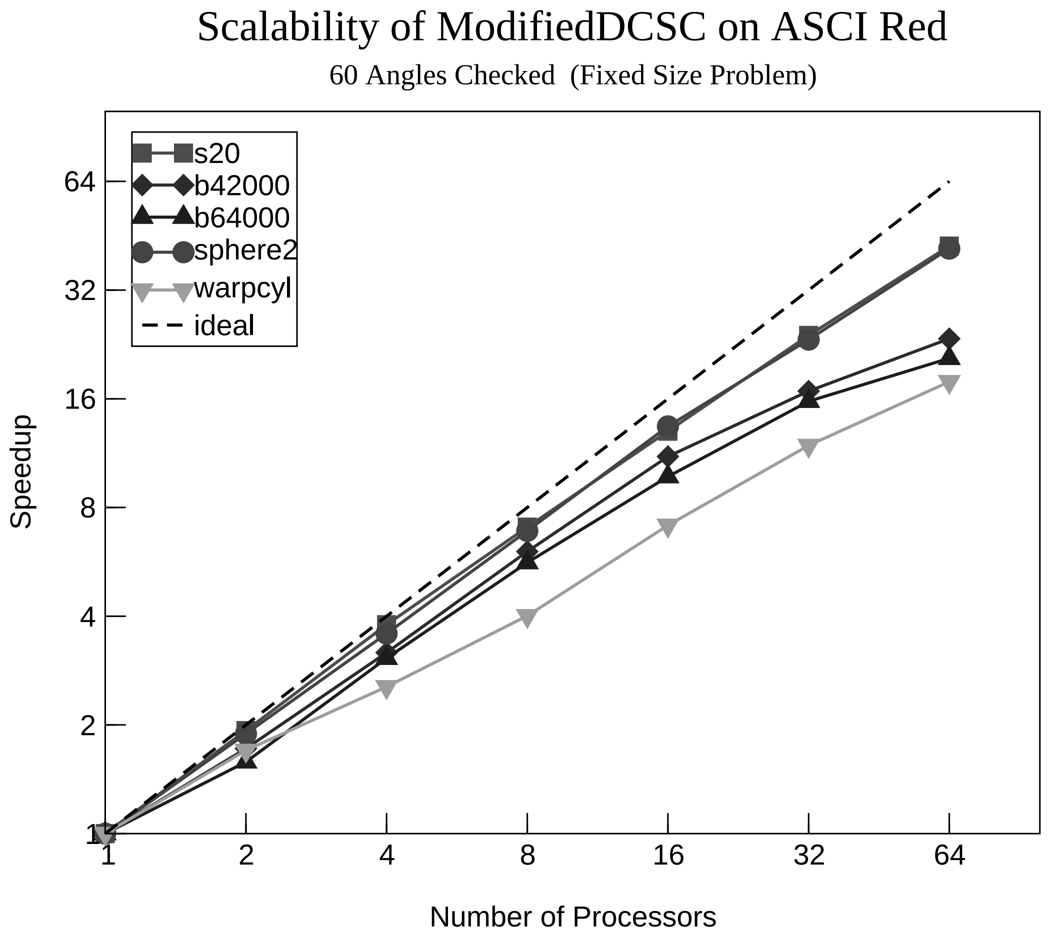


Fig. 12. Scalability of ModifiedDCSC on ASCI Red for various meshes. This graph shows the scalability of ModifiedDCSC searching the DDGs of several meshes of different geometries.

C. Progressively Deformed Meshes

Multi-physics codes operate on meshes which can be slowly deformed at every time-step which would require ModifiedDCSC to also be run every time-step before physics sweeps are attempted. Deformed meshes typically contain more cycles and thus ModifiedDCSC's performance can be reduced. We simulated these changes by progressively increasing the magnitude of deformation of node positions in the mesh. For this purpose we generated a $30 \times 30 \times 30$ brick mesh and moved the corner nodes of the cells randomly. The magnitude of deformation was increased in increments of 10% of the distance to the nearest corner node in a cell. Table IV shows mesh information for this test and the increasing number of SCCs as the magnitude of deformation is increased.

Table IV shows that increasing the displacement of corner nodes corner nodes in mesh cells causes the number of SCCs to increase as well as the average number of nodes contained in each SCC. This implies that as a mesh is increasingly deformed, the connectivity of the resulting DDG is more complex resulting in more SCCs which are larger and contain multiple internal cycles. The larger number of SCCs in these meshes also increases the amount of time ModifiedDCSC requires to compute the full SCC search.

Table V shows the execution time for these meshes, and table VI contains the corresponding speedups. These measurements are for 120 ordinate angles, searching 60 angles due to removal of redundant angles due to ordinate pairing. Figures 13 and 14 show measured scalability on the HP V2200 and ASCI Red.

These results confirm our earlier observation of the impact trim() has on the overall execution of ModifiedDCSC. As the number and density of SCCs increase, ModifiedDCSC benefits less from trimming, resulting in lower performance.

Table IV. Mesh characteristics as a function of deformation.

Mesh	Mesh Size	Deform % Magnitude	# SCCs found	Avg. SCC Size
d_00	$30 \times 30 \times 30$	0	0	0
d_01	$30 \times 30 \times 30$	10	0	0
d_02	$30 \times 30 \times 30$	20	0	0
d_03	$30 \times 30 \times 30$	30	70	4.03
d_04	$30 \times 30 \times 30$	40	899	4.31
d_05	$30 \times 30 \times 30$	50	2701	4.62
d_06	$30 \times 30 \times 30$	60	4825	5.02
d_07	$30 \times 30 \times 30$	70	7120	5.31

Table V. Execution times for deformed meshes on ASCI Red.

Deformed Meshes on ASCI Red							
	Execution Time (seconds)						
Mesh	1	2	4	8	16	32	64
d_00	16.55	8.74	4.51	2.31	1.24	0.66	0.39
d_01	20.88	10.89	5.60	2.85	1.50	0.81	0.45
d_02	20.90	10.89	5.60	2.85	1.50	0.66	0.39
d_03	23.63	12.47	6.55	3.43	1.91	1.11	0.72
d_04	29.00	16.01	8.83	4.80	2.83	1.88	1.36
d_05	34.65	20.76	11.67	6.70	4.02	3.03	2.38
d_06	41.72	24.48	13.62	7.73	5.09	4.39	3.69
d_07	47.53	28.39	16.00	9.53	6.56	5.36	5.65

Table VI. Speedups for deformed meshes on ASCI Red.

	Speedup						
Mesh	1	2	4	8	16	32	64
d_00	1.00	1.89	3.67	7.16	13.35	25.08	42.44
d_01	1.00	1.92	3.73	7.33	13.92	25.78	46.40
d_02	1.00	1.92	3.73	7.33	13.93	31.67	53.59
d_03	1.00	1.89	3.61	6.89	12.37	21.29	32.82
d_04	1.00	1.81	3.28	6.04	10.25	15.43	21.32
d_05	1.00	1.67	2.97	5.17	8.62	11.44	14.56
d_06	1.00	1.70	3.06	5.40	8.20	9.50	11.31
d_07	1.00	1.67	2.97	4.99	7.25	8.87	8.41

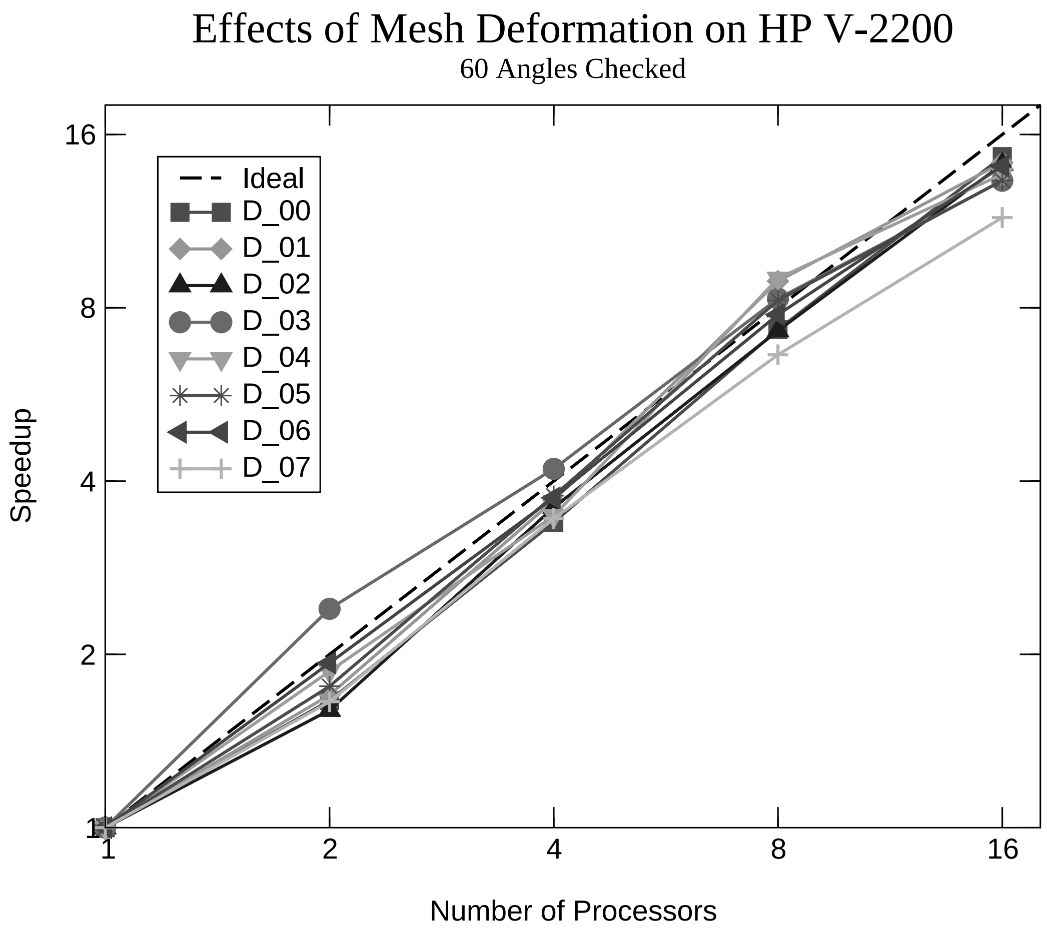


Fig. 13. Scalability of ModifiedDCSC on HP-V2200 for deformed meshes.

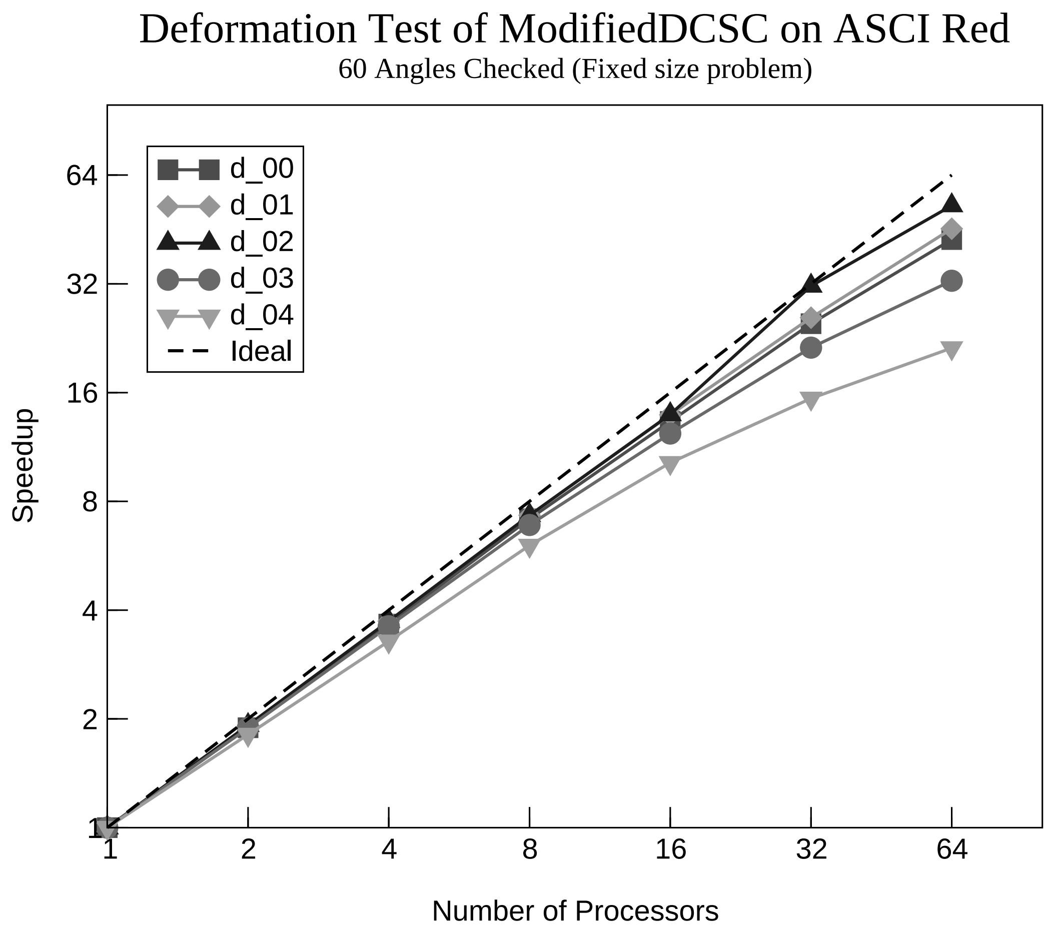


Fig. 14. Scalability of ModifiedDCSC on ASCI Red for deformed meshes.

D. Edge Breaking to Remove SCCs

Recall that in order for a transport sweep to complete we must also give the solver some information about the SCCs that is useful for it. We decided that we would like to break certain edges in the DDG to eliminate the SCCs. Recall that the SCC elimination method we chose to implement involved breaking some edge from each SCC during each recursive step until the whole SCC is eliminated.

Usually when breaking edges from a graph to eliminate SCCs we wish to choose the edges to break based on some criteria. Typically this involves attaching some weight to the edges and trying to maximize or minimize the total weight of cut edges.

In this case the edges we cut correspond to finite element faces and we would like to minimize the error induced from the cut edges on the transport solver. Minimizing the flux through the cell faces will result in a smaller error for the solver, so we can select edges that will minimize this parameter.

CHAPTER V

CONCLUSION

We have presented the ModifiedDCSC algorithm and a parallel implementation that offers a scalable method for detecting the strongly-connected components which arise in sweep calculations for radiation transport.

The addition of the trim step to this algorithm is shown to offer a significant bonus to the execution of the DCSC algorithm. Aggressive trimming reduced the amount of recursion required to find the SCCs in our input graphs. We have also shown that in graphs with few cycles, the addition of trim() allows ModifiedDCSC to complete the SCC search in nearly linear time.

We studied the sensitivity of this algorithm to various characteristics of the input meshes. Not surprisingly, scalability is negatively influenced by the number and density of SCCs of the graph. However, our tests on up to 64 processors of a parallel machine show the overall scalability is reasonable, even for meshes with an artificially large number of SCCs. Moreover, the run times for DCSC are very small compared to actual physics sweeps, making this a useful tool in practice.

ModifiedDCSC can also be easily modified so that it can eliminate SCCs from graphs by cutting certain edges. Our implementation includes this addition to break the SCCs in order to provide information to transport sweeps. This will allow them to sweep an unstructured 3D finite element mesh containing cycles to completion without a deadlock.

This implementation of ModifiedDCSC is now part of a radiation transport package in use at Sandia National Laboratories.

REFERENCES

[1] N. Amato. "Improved processor bounds for parallel algorithms for weighted directed graphs." *Information Processing Letters*, vol. 45, no. 3, pp. 147–152, 1993.

[2] D. Bader. "A practical parallel algorithm for cycle detection in partitioned digraphs." Technical Report AHPCC-TR-99-013, University of New Mexico, Albuquerque, NM, 1999.

[3] R. Cole and U. Vishkin. "Faster optimal parallel prefix sums and list ranking." *Information and Computation*, vol. 81, pp. 334–352, 1989.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* M.I.T. Press, Cambridge, MA, 1990.

[5] L. Fleischer, B. A. Hendrickson, and A. Pinar. "On identifying strongly connected components in parallel." In *IPDPS Workshops*, pp. 505–511, 2000.

[6] H. Gazit and G. L. Miller. "An improved parallel algorithm that computes the BFS numbering of a directed graph." *Information Processing Letters*, vol. 28, no. 2, pp. 61–65, 1988.

[7] P. Gibbons, R. Karp, V. Ramachandran, D. Soroker, and R. E. Tarjan. "Transitive compaction in parallel via branchings." *Journal of Algorithms*, vol. 12, no. 1, pp. 110–125, 1991.

[8] M. Y. Kao and G. E. Shannon. "Linear-processor NC algorithms for planar directed graphs II: Directed spanning trees." Technical Report DUKE–TR–1990–02, Duke University, Durham, NC, 1990.

[9] R. M. Karp and V. Ramachandran. *Parallel Algorithms for Shared-Memory Ma-*

chines, pages 869–941. Jan van Leeuwen, ed., Elsevier Science Publishers B. V., 1990.

[10] L. Lovasz. "Computing ears and branchings in parallel." In *Proc. of 26th Annual IEEE Symposium on Foundations of Computer Science*, Portland, Oregon, pp. 464–467, 1985.

[11] S. J. Plimpton, B. A. Hendrickson, S. P. Burns, and W. C. McLendon III. "Parallel algorithms for radiation transport on unstructured grids". In *Supercomputing 2000 (SC2000)*, Dallas, Texas, November 2000.

[12] J. H. Reif. "Depth-first search is inherently sequential." *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[13] R. E. Tarjan. "Depth first search and linear graph algorithms". *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

VITA

William Clarence McLendon III was born March 20, 1976, in Longview, Texas to William Clarence McLendon Jr. and Sandra Louise McLendon. He graduated with honors from Longview High School in 1994. He received his B.S. in computer science with a minor in electrical engineering at Texas A&M University in 1999. He can be reached via the Department of Computer Science at Texas A&M University / College Station, TX 77843-3112 and via electronic mail at mclendon@cs.tamu.edu.